COMP503 Programming 2/ENSE502 Object-Oriented Applications
Assignment 2 – The Train Seat Booking Application

Due date: Monday, September 18th 2017 at 5pm
This assignment is worth 10% of your final grade
This assignment is marked out of 100

Brief

For this assignment, you will individually develop a Train Seat Booking Application in Java enabling train operators to view the floor grid of a train and to make seat bookings in first or economy class, based on their policies. For a given train journey, the Train Seat Booking Application retrieves its floor grid of seats partitioned into first and economy classes as depicted in the left-hand side of Figure 1. A seat request marks a seat as booked. For example, the first-class seat **3E** is booked and shown in the left-hand side of Figure 1 as a solid circle. This seat is an *aisle* type of seat, since it is next to an aisle. Similarly, *window* type seats are next to the train's windows (such as **5A**), and all other seats are *middle* type seats (such as **9D**).

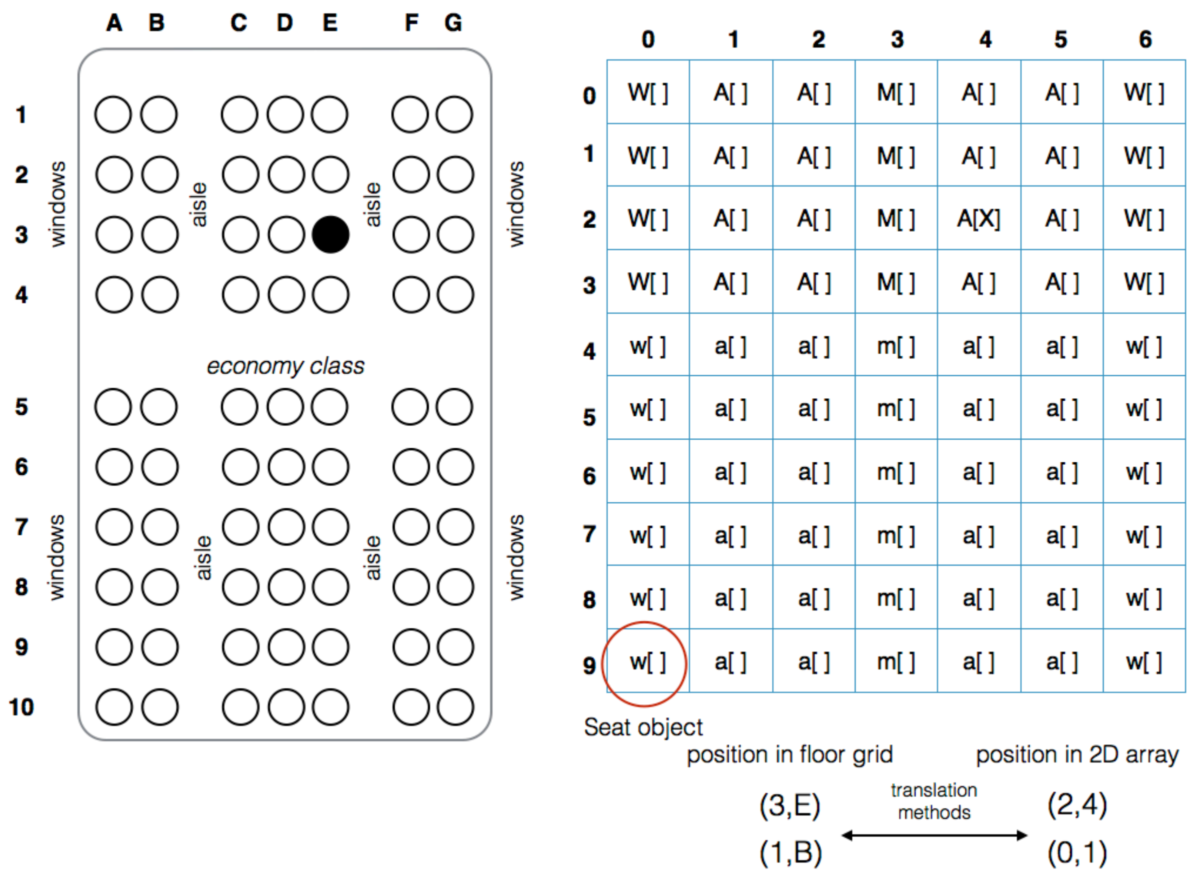

Figure 1: FloorGrid stored as a 2D array of Seat objects

Train Seat Booking Policies

Different train operators have different booking policies if the requested seating is unavailable on a journey. Consider the policies of the following two train operators:

TrainWay

1. **Seat Bookings in First Class**
   A. Find and book a seat in first class that matches the requested seat type
   B. If no such first-class seat with the matching type exists, then find and book any seat in first class
   C. If there are no seats available in first class then instead book a window seat in economy
   D. If there are no seats matching any of these criteria then a booking cannot be made
2. **Seat Bookings in Economy Class**
   A. Find and book a seat in economy class that matches the requested seat type
   B. If no such economy class seat with the matching type exists, then find and book any window seat in first class
   C. If there are no seats matching any of these criteria then a book cannot be made

ChooChoo

1. **Seat Bookings in First Class**
   A. Find and book a seat in first class that matches the requested seat type
   B. If no such first class seat with the matching type exists, then book an entire row in economy class (for extra passenger room)
   C. If there are no seats matching any of these criteria then a booking cannot be made
2. **Seat Bookings in Economy Class**
   A. Find and book a seat in economy class that matches the requested seat type
   B. If there are no seats matching any of these criteria, then a booking cannot be made

Methodology

You will develop a number of classes in Java that implement the Train Seat Booking Application. For full marks, your classes **must** adhere to correct OOP design principles. You

Train Journeys

Design a class to store information relating to a **TrainJourney**. For example, start and destination cities, departure time, journey number and a **FloorGrid** object that stores the floor grid (see below). Consider the data you will need to store to implement the functionality of the Train Seat Booking Application.

Storing Floor Grid and Bookings

In this section you will develop classes to store seat bookings.

Design a **Seat** class which has data to store whether or not a seat is booked or is in first class. The seat has data that stores the type of the seat: AISLE, MIDDLE, WINDOW (Hint: consider using an enumerated type). The **Seat** stores a **SeatPosition** object which contains the row and column of a seat in the floor grid: e.g. **1A**, **6C**, **2G**, forming a floor grid position. Write a toString method which returns a **Seat** representation according to the right-hand side of Figure 1. Write another method that outputs a longer text description of the **Seat**. For example: `Economy class MIDDLE seat at: 9D is not booked.`

Develop an abstract **FloorGrid** class with an abstract method void initialiseFloorGrid(). The class stores a two-dimensional array of **Seat** objects according to the right-hand side of Figure 1, to be initialized by concrete classes (see below). **FloorGrid** contains instance variables that store the number of rows and columns that comprise the matrix and maintains the number of rows situated in first class.

*All instance variables are **protected** in the **FloorGrid** class. You should not write any set methods.*

Instead, it has the following functionality to query the FloorGrid:

1. Provide accessor methods that return the last row (e.g. 10) and last column (e.g. the character 'G') in the floor grid obejct.

2. Seat getSeat(int, char) returns the seat in the specified floor grid position (Refer to Figure 1).

3. getLeft (Seat)/getRight(Seat), returns a seat to the left or right of the input Seat. (Hint: use SeatPosition to find the seat in the 2D array). If the seat does not exist, return null.

4. Seat queryAvailableEconomySeat(SeatType) returns a seat in economy that has the matching SeatType and is not already booked. If all these types of seats are booked, return any seat in economy. If all seats are booked, return null.

5. Seat queryAvailableFirstClassSeat(SeatType) is the same, but only searches first class.

6. toString returns a string containing a text representation of the **FloorGrid** similar to the right-hand side of Figure 1.

Develop the following two classes that extend **FloorGrid**:

PetiteFloorGrid has a default constructor initialising the number of rows to 10, the number of columns to 7 and the number of first class rows to 4. The constructor initialises the **seats** array accordingly. The constructor calls the **initialiseFloorGrid** method. Write an algorithm in **initialiseFloorGrid** to populate the **seats** array according to Figure 2.

GrandeFloorGrid is a concrete class extending the **FloorGrid** class has a default constructor initialising the number of rows to 12, the number of columns to 9 and the number of first class rows to 6. The constructor initialises the **seats** array accordingly. The constructor calls the **initialiseFloorGrid** method. Write an algorithm in **initialiseFloorGrid** to populate the **seats** array according to Figure 2.

Note: **initialiseFloorGrid** instantiates each **Seat** object stored in the seats array with the appropriate SeatPosition object by translating the array indices to a **SeatPosition**.

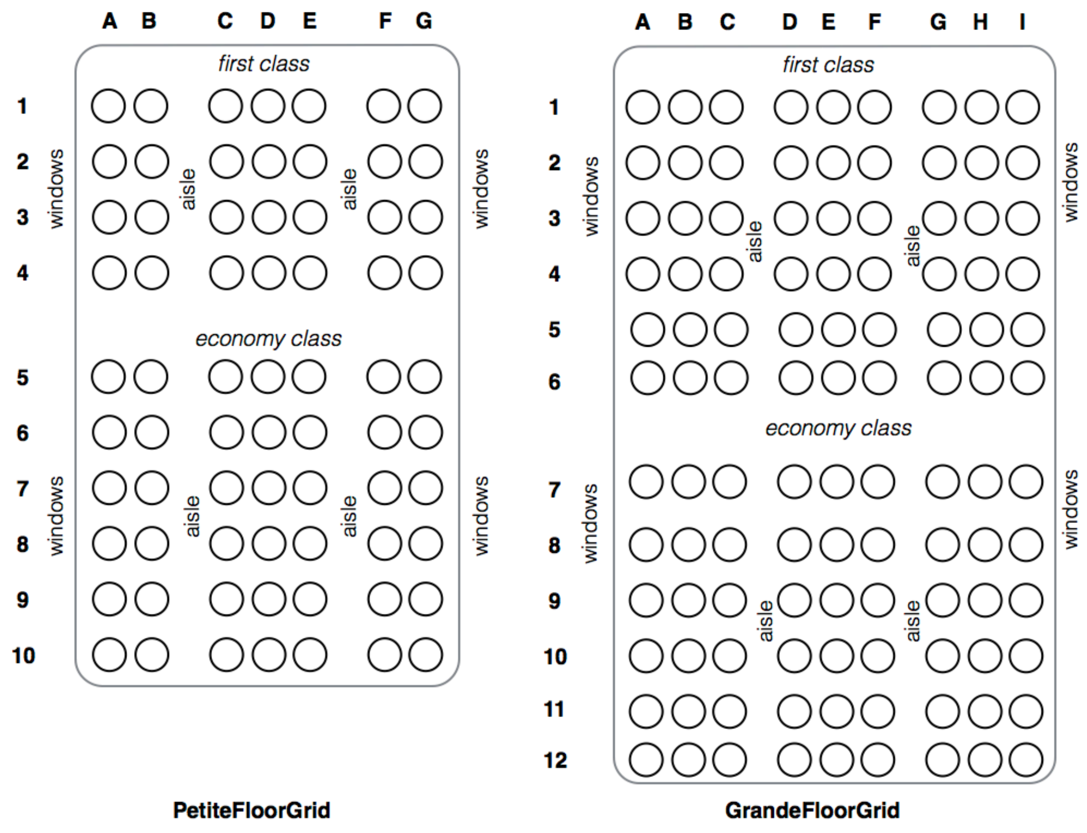

Figure 2: FloorGrid subclasses PetiteFloorGrid and GrandeFloorGrid

Booking a Train Seat

Train operators query a train's floor grid to determine the availability of seats. Create an abstract class **TrainOperator**. The class contains a descriptive name of the train operator and get/set methods. Define a toString method that prints out a nice representation of the object. This class has two *abstract* methods:

- `Seat bookFirstClass(TrainJourney, SeatType)`
- `Seat bookEconomy(TrainJourney, SeatType)`

TrainOperator is extended by two concrete classes **TrainWay** and **ChooChoo** and each class implements the abstract methods *according to the train operator's seat booking policies.*

Program Interaction

Develop an application class called **TrainSeatBookingApplication** which interacts with the core functionality of the Train Seat Booking Application. Your program should involve multiple train journeys to select from, different seating configurations and demonstrate both train operators seat booking policies. You may consider writing additional methods to simulate when economy or first class is completely booked out.

Marking Scheme

Criteria:	Weight:	Grade A Range 100 ≥ x ≥ 80%	Grade B Range 80 > x ≥ 65%	Grade C Range 65 > x ≥ 50%	Grade D Range 50 > x ≥ 0%
Functionality of FloorGrid classes	**30%**	OOP paradigm consistently used for implementation of all FloorGrid functionality.	Inconsistent use of OOP paradigm but correct implementation of FloorGrid functionality	Incorrect FloorGrid functionality and poor use of OOP paradigm	Absent FloorGrid functionality or code does not compile
Functionality of TrainOperator classes	**15%**	OOP paradigm consistently used for implementation of all Train Operator functionality.	Inconsistent use of OOP paradigm but correct implementation of TrainOperator functionality	Some basic functionality of TrainOperator classes/poor usage of abstract and concrete classes	Absent functionality of TrainOperator classes or code does not compile.
Program's Runtime Demonstration: -Uniqueness -Purpose -Context -Interactive	**25%**	The object-oriented program is unique, purposeful and provides an elegant implementation of the Train Seat Booking Application. Program is interactive.	The object-oriented program is unique, purposeful. Reasonable implementation of the Train Seat Booking Application. Program is interactive.	The object-oriented program features an incomplete demonstration of the Train Seat Booking Application Program is not interactive.	Absent functionality of Train Seat Booking Application or code does not run after compiling
Code Quality: -Whitespace -Naming -Reuse -Modularity -Encapsulation	**15%**	Whitespace is comprehensively consistent. All naming is sensible and meaningful. Code reuse is present. Code is modular. Code is well encapsulated.	Whitespace is comprehensively consistent. Majority of naming is sensible. Code is modular. Code is encapsulated.	Whitespace is comprehensively consistent. Code has some modularity. Code has some encapsulation.	Whitespace is inconsistent and hence code is difficult to read.
Documentation Standards: -Algorithms Commented -Javadoc	**15%**	Entire codebase has comprehensive Javadoc commenting. Algorithms are well commented.	Majority of the codebase features Javadoc commenting. Majority of algorithms are commented.	Some Javadoc comments present. Some algorithms are commented.	No Javadoc comments present.

Authenticity

Remember, it is unacceptable to hand in any code which you have previously obtained credit (in any paper), and all work submitted must be unique and your own! We use automated methods to detect academic integrity breaches.

Submission Instructions

Submit the following documents as an archive `.zip` file of your documents on Blackboard before the deadline:

- Your source code (`.java` files).
- Sample console output demonstrating your program in use (`.txt` file)

Zip structure and file naming requirements. Please ensure your submission matches the following:

```
🗐 lastname-firstname-studentid.zip
    🗎 *.java
    🗎 studentid-console-sample.txt
```

Replace the underlined text with your personal details.

Late submissions will receive a grade of 0

An extension will only be considered with a Special Consideration Form approved by the School Registrar. These forms are accessible via Blackboard.

You will receive your marked assignment via Blackboard. Please look over your entire assignment to make sure that it has been marked correctly. If you have any concerns, you must raise them with the lecturer. You have **one week** to raise any concerns regarding your mark. After that time, your mark cannot be changed.

Do not email the lecturer because you do not like your mark. Only go if you feel something has been marked incorrectly.